# Isomorphic, Sparse MPI-like Collective Communication Operations for Parallel Stencil Computations*

Jesper Larsson Träff
traff@par.tuwien.ac.at

Felix Donatus Lübbe
luebbe@par.tuwien.ac.at

Antoine Rougier
rougier@par.tuwien.ac.at

Sascha Hunold
hunold@par.tuwien.ac.at

Vienna University of Technology (TU Wien)
Faculty of Informatics, Institute of Information Systems
Research Group Parallel Computing

## ABSTRACT

We propose a specification and discuss implementations of collective operations for parallel stencil-like computations that are not supported well by the current MPI 3.1 neighborhood collectives. In our *isomorphic, sparse collectives* all processes partaking in the communication operation use similar neighborhoods of processes with which to exchange data. Our interface assumes the $p$ processes to be arranged in a $d$-dimensional torus (mesh) over which neighborhoods are specified per process by identical lists of relative coordinates. This extends significantly on the functionality for Cartesian communicators, and is a much lighter mechanism than distributed graph topologies. It allows for fast, local computation of communication schedules, and can be used in more dynamic contexts than current MPI functionality. We sketch three algorithms for neighborhoods with $s$ source and target neighbors, namely a) a direct algorithm taking $s$ communication rounds, b) a message-combining algorithm that communicates only along torus coordinates, and c) a message-combining algorithm using between $\lceil \log s \rceil$ and $\lceil \log p \rceil$ communication rounds. Our concrete interface has been implemented using the direct algorithm a). We benchmark our implementations and compare to the MPI neighborhood collectives. We demonstrate significant advantages in set-up times, and comparable communication times. Finally, we use our isomorphic, sparse collectives to implement a stencil computation with a deep halo, and discuss derived datatypes required for this application.

*This work was co-funded by the European Commission through the EPiGRAM project (grant agreement no. 610598) and supported by the Austrian FWF project "Verifying self-consistent MPI performance guidelines" (P25530).



*EuroMPI '15, September 21-23, 2015, Bordeaux , France*

 ISBN 978-1-4503-3795-3/15/09…$15.00

DOI: http://dx.doi.org/10.1145/2802658.2802663

## 1. INTRODUCTION

Collective communication operations are convenient for expressing and implementing parallel computations. Global collective operations as found in MPI [16] require the participation of all processes in a given communication domain (communicator), and processes provide input to all other processes in a similar fashion. A different type of collective operation allows each participating process to interact with only a small neighborhood of processes, a pattern for instance found in stencil- and halo-type computations. The MPI 3.1 standard [16] provides *sparse* or *neighborhood collectives* of either the alltoall (or scatter) type where each process has possibly different data for each of its neighbors, or the allgather type, where each process communicates the same data to all of its neighbors. Neighborhoods are described either by completely general *communication graphs* [9, 16], or as restricted, Cartesian neighborhoods. The intention is that neighborhood collective operations can be used to implement nearest-neighbor stencil-like algorithms in a more structured way with the low-level concerns of finding an efficient communication schedule being handled by the MPI library [3, Section 2.3].

The MPI interface for sparse collective communication is conceptually heavy, and requires setting up an implicit communication neighborhood by a generic mechanism. The generality of the mechanism makes optimization by the MPI library in specific, natural application cases difficult. We propose a more lightweight interface based on the observation that stencil-computations are formulated relative to mesh, tori or other highly structured underlying topologies. Process neighborhoods are then specified by lists of relative offsets, similar to the way neighborhoods are defined for cellular automata [20]. Our proposal in a natural way extends the Cartesian topologies of MPI, which are of severely limited expressivity. By explicitly asserting process neighborhoods to be *isomorphic*, the MPI library can immediately use structured algorithms for the communication and more efficiently perform non-trivial optimizations without the need for global communication. We outline three such algorithms assuming $k$-ported, bidirectional communication. The first algorithm assumes a fully-connected network, is optimal in the total volume of data communicated, and uses $\lceil s/k \rceil$ communication rounds for neighborhoods with $s$ neighbors. The second

algorithm assumes that the virtual mesh or torus topology has been efficiently mapped to the actual communication network, and communicates only along the mesh dimensions. It uses message combining and requires a smaller number of rounds, depending on the structure of the neighborhood. The third algorithm also assumes a fully connected network, and uses message combining to achieve between $\lceil \log_{k+1} s \rceil$ and $\lceil \log_{k+1} p \rceil$ communication rounds where $p$ is the number of processes. The exact number of rounds depends on the neighborhood structure.

Our new interface has been implemented on top of MPI[1]. We use it to implement standard 5- and 9-point stencil communication patterns, and compare to implementations using the current MPI 3.1 neighborhood collectives. We also describe a benchmark for systematically estimating the performance of structured uses of neighborhood collectives. We present first results with different MPI libraries on a small 36-node InfiniBand cluster, and demonstrate significant cost savings in setting up the neighborhoods with our interface over the MPI 3.1 distributed graph interfaces, with comparable, sometimes better communication performance.

Neighborhood collectives were added to MPI with the MPI 3.0 standard in 2012, and while current, open-source MPI implementations support the operations, there is still not much work on theoretical and practical optimizations for the neighborhood collectives. Hoefler and Schneider discuss general principles [10], and use the `MPI_Info` parameter to define different levels of persistence: communication topology, message sizes, and communication buffers, to provide for better communication scheduling and low-level RDMA optimizations. Significant benefits are shown for 4D stencil computations for smaller message sizes; also other real-world applications can benefit from the optimized implementations. Many of these optimizations are considerably easier with the asserted, global information that processes have isomorphic neighborhoods. We illustrate this claim with some examples. Kumar et al. [15] demonstrate the usefulness of the neighborhood collectives for improving applications where each process communicates with only a subset of the other processes, such as the three-dimensional Fast Fourier Transform. To perform the neighborhood communications, they use multisend operations from the low-level messaging library of the Blue Gene/P. They show that neighborhood collectives can facilitate communication-computation overlap, and can outperform point-to-point communication and classic, global MPI collectives. Kandalla et al. [14] investigate the use of non-blocking neighborhood collectives to reduce communication overhead in irregular graph algorithms. They show how generic collective algorithms can be efficiently implemented by using multiple calls of neighborhood collectives, and divide the original communication graph into sub-graphs with configurable numbers of neighbors, which makes it possible to achieve better communication-computation overlap and scalability.

None of the commonly used MPI benchmarks, like for instance SKaMPI [18], `mpptest` [4], OSU Micro-Benchmarks [17], or Intel MPI Benchmarks [13], have settings for benchmarking the neighborhood collectives. LibNBC [8] implements (nonblocking) MPI neighborhood collectives, which can be benchmarked using the corresponding NBCBench [11].

[1]A library is available at www.par.tuwien.ac.at/Downloads/TUWMPI/tuwisosparse.tgz.

## 2. MOTIVATING EXAMPLES

We first consider two standard stencil problems that can be solved using neighborhood collective operations. For visual clarity, we consider only the two-dimensional versions. Figure 1 illustrates the communication pattern for a 5-point stencil and a 9-point stencil computation both with a *halo* of depth $k$, $k \geq 1$. The MPI processes are organized in a two-dimensional mesh and each operates on a local, square matrix of order $n$. Each matrix element is updated by a reduction (e.g., average) over its 5 or 9 neighbors (including itself) for which information from the neighboring matrices is needed. This update computation is iterated until some criterion is met, see [5] or any other standard textbook. The communication to be done for each process includes sending and receiving data on the borders of the matrix to either 4 or 8 neighbors, and can be accomplished with a sparse, neighborhood collective operation like `MPI_Neighbor_alltoallv`. The neighborhood of 4 processes is sometimes termed the *von Neumann neighborhood*, and the full 8 process neighborhood the *Moore neighborhood* [20]. The important observation is that the structure of the neighborhood is the same for all processes; only processes sitting at the borders of the mesh have virtual neighbors with which they do no communication. The whole exchange can be done in a collective fashion with all processes participating and all neighbors being handled in the operation. We will call neighborhood collective communication under these conditions *isomorphic, sparse collective communication*. With the use of a deeper halo of depth $k$, $k > 1$, synchronization and data exchange need to be done only at every $k$th iteration of the stencil computation, but with a roughly $k$ times larger communication volume; finding the best tradeoff is not our concern here. An important observation, however, is that when the halo-depth is larger than one, both stencil patterns need to exchange information with 8 neighboring processes, namely the four process neighbors along the mesh coordinates, as well as the four corner neighbors. The data to be communicated in the up-down directions have a row-wise layout, whereas columns have to be exchanged in the left-right directions. The communication volume in the four principal directions is $nk$. The corners have different layouts in the two cases. The 5-point stencil needs to exchange triangular submatrices of $k(k-1)/2$ elements, whereas full, rectangular submatrices of $k^2$ elements are exchanged in the 9-point stencil pattern. For implementation in MPI, both row- and column-wise data layouts can be described with derived (user-defined) datatypes. This makes it possible to implement the exchange in a *zero-copy* fashion [7, 24], which means that data are accessed in-place by the communication operations with no need for explicit copying into dedicated communication buffers. Although there is no MPI derived datatype constructor for triangular layouts, all the different layouts can easily be described by MPI derived datatypes. Note finally, that for the corners partly overlapping buffers are sent to different neighbors.

For the sparse collective communication operations to enable zero-copy implementations with derived datatypes they must support exchange of differently structured data along the $d$ mesh-dimensions as well as for the corners. Therefore, sparse collective operations of the `-w`-variety, where each neighbor is given its own, possibly different, MPI datatype in addition to its own count and displacement would seem to be the most useful ones. MPI provides the `MPI_Neighbor_alltoallw` collective and its non-blocking counterpart, either

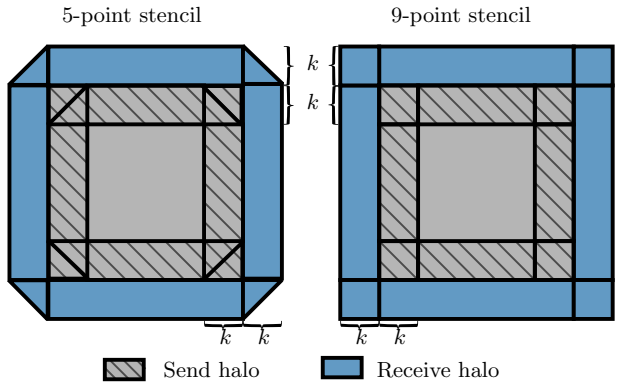


**Figure 1: The communication patterns for 5- and 9-point stencil computations with halo of depth $k$. Rows, columns, and corners all have different data layouts. The structure of the corners is different in the two cases.**

of which is the operation fitting here. For the global collective operations, this is not a scalable interface specification [1] (see [23] for a possible solution); but this is not likely to be a problem in the sparse case where the number of neighbors is very small (constant) relative to the total number of processes. A *triangular vector datatype* would be convenient for the corner exchanges, and we discuss this briefly later in the paper.

# 3. SPARSE COLLECTIVE COMMUNICATION IN MPI

Both 5-point and 9-point stencil communication can be implemented using the MPI neighborhood collectives; indeed, these types of communication patterns motivated the inclusion of neighborhood collectives in MPI 3.0. In both cases, processes are (implicitly) assumed to be organized in a Cartesian mesh: neighborhoods are defined relative to this virtual topology. Whether the virtual topology corresponds or has been mapped well to the physical network is a different matter, and for the moment of no concern.

With the 5-point stencil and a halo of size 1, communication is along the mesh dimensions. In this case, an MPI Cartesian communicator suffices, since the neighborhood implicitly defined by MPI for the neighborhood collective operations consists of exactly the immediate, radius-one von Neumann neighbors. The order of these neighbors is defined by the MPI standard [16, Section 7.6], which is important for addressing the communication buffers used in the neighborhood collectives. If the Cartesian topology is indeed a mesh and not a torus, some neighbors of some processes are `MPI_PROC_NULL`; such neighbors will simply be ignored in the communication and the associated (part of the) communication buffer skipped. For the 9-point, but also the 5-point stencil case with a non-trivial, deeper halo, there is also communication along diagonals in the mesh. This cannot be expressed as collective communication over a Cartesian communicator, because of its implicit, fixed neighborhood. To implement this pattern, a general graph communicator must be set up, for instance with the `MPI_Dist_graph_create_adjacent` constructor. A slight drawback is that the user must translate manually from relative neighbor offsets into actual ranks, and that non-existing mesh neighbors for processes at the mesh borders must be explicitly removed, since the graph constructors accept only actual processes as source and destination parameters and not `MPI_PROC_NULL`. A more severe drawback is that the information that the neighborhood is highly structured (isomorphic) is lost to the MPI library; any optimizations that might apply on such structured neighborhoods would require an expensive analysis of the distributed graph to rediscover the regular structure.

These observations and concerns motivate a simpler, more uniformly structured interface for sparse collective communication in meshes and tori.

# 4. ISOMORPHIC, SPARSE COLLECTIVE COMMUNICATION

We now define precisely what we mean by *isomorphic, sparse collective communication*. Isomorphic communication patterns are defined relative to a given, structured organization of the MPI processes. Let $p$ be the number of MPI processes, and assume at first that they are organized in a $d$-dimensional torus with dimension sizes $p_0, p_1, \ldots, p_{d-1}$ and $\Pi_{i=0}^{d-1} p_i = p$. Each process $X$ is identified by a coordinate $(x_0, x_1, \ldots x_{d-1})$ with $0 \leq x_i < p_i$ for $i = 0, \ldots, d-1$.

A (sparse) *s-neighborhood* for a process is a collection of $s$ processes to which the process shall *send* data. The collection is given as a(n ordered) sequence of $s$ *relative coordinate vectors* $C^0, C^1, \ldots C^{s-1}$. Each $C^i$ has the form $(c_0^i, c_1^i, \ldots, c_{d-1}^i)$ for arbitrary integer offsets $c_j^i$ (positive or negative). A set of identical $s$-neighborhoods for a set of processes is said to be *isomorphic*. An *isomorphic, sparse collective operation* is a collective operation over $p$ processes with isomorphic neighborhoods. Note that an $s$-neighborhood is allowed to have repetitions of some relative coordinate. A process is a neighbor of itself if relative coordinate $(0, 0, \ldots, 0)$ is in the $s$-neighborhood.

Each process $(x_0, x_1, \ldots, x_{d-1})$ with $s$-neighborhood $C^0$, $C^1, \ldots, C^{s-1}$ shall send data to the $s$ *target processes* $((x_0 + c_0^i) \bmod p_0, (x_1 + c_1^i) \bmod p_1, \ldots, (x_{d-1} + c_{d-1}^i) \bmod p_{d-1})$. Since neighborhoods are isomorphic, it follows that the process will need to receive data from $s$ *source processes* $((x_0 - c_0^i) \bmod p_0, (x_1 - c_1^i) \bmod p_1, \ldots, (x_{d-1} - c_{d-1}^i) \bmod p_{d-1})$.

We define the following isomorphic collective communication problems. The unit of communication is called a *block* or a *vector*:

- (all)gather: each process sends the same block to each of its target processes, and, per symmetry, receives a block from each of its source processes.
- alltoall (scatter): each process sends a personalized block of data to each target process, and, per symmetry, receives a block from each of its source processes.
- (commutative) reduction: each process receives a vector from each of its source processes and performs a commutative reduction operation over the received vectors. Each process contributes the same vector to each of its targets.
- (commutative) scatter-reduction: as above, except that each process contributes a possibly different vector to each of its targets

The allgather and alltoall problems can be either *regular* or *irregular*. The former means that all data blocks have the same size (number of elements), whereas the latter allows different block sizes. For the reduction problems, all vectors must have the same size.

The definition can be extended to incomplete tori where processes along some dimensions may be organized as linear arrays and not rings. If there is an incomplete dimension $j$ (non-periodic, in MPI terminology) such that the coordinate $x_j+c_j$ of the vector $X+C$ is either negative, or if $x_j+c_j \geq p_j$, then there will be no communication with relative neighbor $C$ of $X$.

# 5. ALGORITHMS

We now sketch algorithms for realizing isomorphic, sparse communication operations, taking advantage of the additional information that process neighborhoods are isomorphic. We consider primarily the alltoall and allgather problems. We assume a communication network supporting $k$-ported, $k \geq 1$, bidirectional communication: each process can send data to at most $k$ other processes and simultaneously receive data from at most $k$ (other) processes at the same time. As above, $s$ is the number of neighbors in the neighborhoods.

We are interested in finding correct, deadlock-free algorithms for the sparse communication problems with either of the following properties:

1. No message-combining and $\lceil s/k \rceil$ communication rounds, assuming a fully connected network.
2. Allowing messages to be combined and communication restricted to the dimensions of the torus, smallest possible number of communication rounds.
3. Allowing messages to be combined, assuming a fully connected network, smallest possible number of communication rounds. Note that $\lceil \log_{k+1} s \rceil$ rounds is a lower bound [2].

We would also be interested in algorithms for hierarchical systems which perform as few simultaneous communication operations between nodes as possible.

## 5.1 Algorithm 1: Direct communication

It is easy to give an algorithm for the first case when $k = 1$. The neighborhood for each process consists of $s$ target and $s$ source processes. For the collective communication, each process $X$ in the torus simply performs $s$ send-receive rounds, in round $i$ sending to process $X + C^i$, and receiving from process $X - C^i$. Deadlock-freedom of this simple scheme is an immediate consequence of the neighborhoods being isomorphic: in the round where $X$ receives from $X - C^i$, process $X - C^i$ sends to process $(X - C^i) + C^i = X$. For regular problems, the algorithm has no idle time, since all processes are sending and receiving data of the same size in each round and are thus fully occupied throughout the $s$ rounds. This also holds for irregular problems, if the $i$th target and source blocks have the same size.

For $k > 1$ we instead do the communication in $\lceil s/k \rceil$ rounds. For regular problems there is again no idle time. For irregular problems, we can sort the neighborhood according to the target data sizes, and try to achieve the least possible difference between the $k$ blocks sent in a communication round.

## 5.2 Algorithm 2: Message-combining along dimensions

The next two algorithms reduce the number of communication rounds by combining messages. Assume first that it is preferable to communicate only along the torus dimensions. For any neighbor $C^i = (c^i_0, c^i_1, \ldots, c^i_{d-1})$ in the $s$-neighborhood, write $C^i$ uniquely as $(c^i_0, c^i_1, \ldots, c^i_{d-1}) = (c^i_0, 0, \ldots, 0) + (0, c^i_1, \ldots, 0) + (0, 0, \ldots, c^i_{d-1})$, e.g., as a sum of basis vectors. If we assume that each of the $d$ basis vectors $(0, \ldots, c^i_j, \ldots, 0)$ are also in the $s$-neighborhood, then the block to neighbor $C$ can be sent in $d$ rounds by sending along the coordinate dimensions and combining it with other blocks. In round $j$ the block is forwarded from process $X + (0, \ldots, c^i_j, 0, \ldots, 0)$ to $X + (0, \ldots, 0, c^i_{j+1}, \ldots, 0)$. Again, since neighborhoods are isomorphic, all processes will do communication in the same relative direction in each of the $d$ rounds, so the scheme is deadlock free.

To check whether all required basis vectors for relative neighbor $C$ are in the $s$-neighborhood, we first scan through the neighbors and put the basis vectors in buckets, one per dimension. If there are more than one basis vector on some dimension, the corresponding bucket is sorted. Computation of the schedule can thus be done in $O(sd \log s)$.

To illustrate the savings possible by message-combining along dimensions, consider a full $d$-dimensional Moore neighborhood of radius 1 with $3^d - 1$ neighbors. With message combining along the dimensions, this can be handled in $2d$ 1-ported communication rounds.

Many variations of this scheme are possible, for instance depending on whether it shall be permitted to route through processes that are not part of the neighborhood; the minimum number of communication rounds depends on the exact structure of the $s$-neighborhood.

## 5.3 Algorithm 3: Message-combining with minimal number of rounds

Here, we make the observation that the well-known message-combining algorithm of Bruck et al. [2] can be adapted to sparse communication in isomorphic $s$-neighborhoods. This leads to an algorithm taking $\lceil \log_{k+1} s \rceil$ communication rounds in the best case, which is also the smallest possible.

The intuition is as follows. Assume that the $p$ processes are arranged in a ring, and that process $i$ has $s$ source neighbors $i-1, i-2, \ldots, i-s$ and $s$ target neighbors $i+1, i+2, \ldots, i+s$. The processes do $\lceil \log s \rceil$ communication rounds. In round $j$, process $i$ sends blocks forward to process $i + 2^j$, namely the blocks destined to some target neighbor $i + k$ where $k = 2^j + x$ and $x$ is a sum of powers of two excluding $2^j$. Since all processes follow the same communication pattern, the scheme will not deadlock. Per round, each process sends and receives $\lceil s/2 \rceil$ blocks. This works for any linear arrangement of processes, so if the $s$-neighborhood happens to be a linear embedding in the $d$-dimensional torus, this number of rounds can be achieved. This can be checked in $O(sd)$ time steps.

For general neighborhoods we can solve the sparse problem as a global, irregular alltoall problem. We arrange the processes in a ring by process rank. Each process' $s$-neighborhood determines exactly from which processes it shall receive and to which processes it shall send. Since the neighborhoods are isomorphic, each process will in each round have to send and receive the same number of blocks. The actual number of communication rounds required is $\lceil \log_{k+1} r \rceil$, where $r$ is the longest distance from a process

to a neighbor in the ring. Depending on the structure of the neighborhood, better linear embeddings with a smaller value of $r$ may exist. Another way to exploit the Bruck algorithm would be to decompose the $s$-neighborhood into a set of linearly ordered neighborhoods, and process them simultaneously using the available communication ports.

# 6. AN INTERFACE

We now present a simple, light-weight interface for isomorphic sparse collective communication in arbitrary Cartesian communicators. The interface is intended to allow faster setup than distributed graph communicators, and to allow local computations of good communication schedules. It consists of some convenience functions, the collective operations for setting up neighborhoods, and finally the sparse collective communication operations.

For a $d$-dimensional Cartesian MPI communicator `cartcomm`, processes are identified by their rank as well as by their coordinates. Coordinates and relative coordinate vectors are represented as flat, $d$-dimensional integer arrays. The following three translation functions are useful for navigating in sparse neighborhoods and assign ranks to relative coordinate offsets and vice versa.

```
// compute absolute rank relative to caller
Cart_relative_rank(MPI_Comm cartcomm, int relative[],
                   int *rank)

// compute relative coordinate from caller
Cart_relative_coord(MPI_Comm cartcomm, int rank,
                    int relative[])

// generalized shift in relative direction
Cart_relative_shift(MPI_Comm cartcomm, int relative[],
                    int *source, int *target)
```

For Cartesian tori (in MPI terms: all dimensions being periodic) all translations are well-defined, in the sense that there is a corresponding torus coordinate for any relative coordinate vector. If some dimension is not periodic, adding the relative coordinate of that dimension may not be in range (either $r_i + c_i < 0$ or $r_i + c_i \geq p_i$), and the translation is therefore undefined. Our interface returns `MPI_PROC_NULL` for such cases. These functions can trivially be implemented with existing MPI functionality for translating between ranks and coordinates in the Cartesian communicator.

Our isomorphic neighborhood constructor must be called on a Cartesian topology and attaches an $s$-neighborhood of relative coordinates. The call is formally collective, and the requirement is that all MPI processes call with *exactly the same local neighborhood*.

```
Iso_neighborhood_create(MPI_Comm cartcomm, int s,
    int relative_coordinates[], MPI_Comm *isocomm)
```

The array `relative_coordinates` is a flattened list of $s$ $d$-tuples. The call is collective because a new, dedicated communicator has to be created. Our actual implementation does not create a new communicator, but just attaches the necessary information (lists of source and target ranks) to the given Cartesian communicator using an attribute with a reserved (generated) key value. For each MPI process, the list of relative coordinates translates into a list of absolute ranks that is used internally for our sparse collective operations. The order of the list is important, both because it ensures deadlock freedom, and because it determines the order of the communication buffers used in the collective operations.

Isomorphic neighborhoods are set up on top of Cartesian communicators. Thus, we assume that process remapping to fit the Cartesian structure onto the underlying network has been performed by the time the Cartesian communicator was created. We note that the MPI 3.1 interface does not allow to specify weights for the communication links in Cartesian topologies, which in this respect differ from the general graph topologies. For neighborhoods with neighbors that are not direct neighbors in the Cartesian topology as defined by MPI, weighted reorderings could possibly lead to better mappings. We could have defined our `Iso_neighborhood_create` to associate a weight with each neighbor (list of weights), have a `reorder` flag and also take `MPI_Info`, and thus have the possibility to perform an even better mapping for irregular sparse collective operations, but for simplicity we chose not to.

Query functions are defined in analogy with the distributed graph interface of the MPI 3.1 standard. The first function returns the size of the $s$-neighborhood of the calling process, as well as the in- and out-degree of the calling process in the implicit communication graph of neighbors excluding any `MPI_PROC_NULL` ones. The second function returns the absolute ranks of the first `max_s` target and source processes of the neighborhood of the calling process as defined, including possible `MPI_PROC_NULL` neighbors. The third function excludes `MPI_PROC_NULL` neighbors, and the lists returned can therefore be used immediately as input to the distributed graph creation function `MPI_Dist_graph_create_adjacent`

```
Iso_neighborhood_count(MPI_Comm isocomm, int *s,
                       int *indegree, int *outdegree);

Iso_neighborhood_get(MPI_Comm isocomm, int max_s,
    int sources[], int destinations[])

Iso_neighborhood_graph_get(MPI_Comm isocomm, int max_s,
    int sources[], int destinations[])

Cart_neighborhood_count(MPI_comm cartcomm, int *s,
                        int *indegree, int *outdegree);

Cart_neighborhood_get(MPI_Comm cartcomm, int max_s,
     int sources[], int destinations[])

Cart_neighborhood_graph_get(MPI_Comm cartcomm, int max_s,
     int sources[], int destinations[])
```

For convenience we provide functions to return in the same way the predefined neighbors of a Cartesian communicator.

Isomorphic, sparse collective operations can now be used on communicators with attached $s$-neighborhoods. The irregular alltoall function looks like this:

```
Iso_neighbor_alltoallw(void *sendbuf, int sendcount[],
    MPI_Aint senddisp[], MPI_Datatype sendtype[],
    void *recvbuf, int recvcount[],
    MPI_Aint recvdisp[], MPI_Datatype recvtype[],
    MPI_Comm isocomm)
```

The $i$th sendbuffer block is defined by count, displacement (relative to the `sendbuf` address) and datatype. This block will be used for the data transferred to the $i$th neighbor in the isomorphic neighborhood. The receive buffer blocks are handled likewise. As discussed in Section 2, for efficient (zero-copy) implementations it is necessary that each block can be given its own datatype. Therefore, our interface is of the `-w`-variety, although this is not particularly elegant or space-efficient. Recall from the stencil examples that partly overlapping data may have to be sent to different processes. For non-periodic mesh topologies, there might not

be an actual neighbor for the $i$th relative coordinate. In such cases, the arguments for the $i$th buffer are not significant. We also provide interface definitions for `Iso_neighbor_alltoall` and `Iso_neighbor_alltoallv`, and for three similar `Iso_neighbor_allgather` functions.

Reduction operations that may be useful for computing averages over neighborhoods are provided in two flavors as defined in Section 4. The interface for the scatter-reduce operation looks as follows:

```
int Iso_neighbor_scatter_reduce(void *sendbuf,
    int sendcount[], MPI_Aint senddisp[],
    MPI_Datatype sendtype[],
    void *recvbuf, int count, MPI_Datatype datatype,
    MPI_Op commutative_op, MPI_Comm isocomm)
```

A different vector can be provided to each target neighbor. A vector with the same number of elements is received from each source neighbor and reduced, assuming the MPI operation provided is commutative. By commutativity, the computed result is independent of the order of neighbors in the $s$-neighborhood. Note that a process does not necessarily perform a reduction with a vector in the send buffer. This will happen only if the processes are neighbors of themselves. Therefore, the `MPI_IN_PLACE` argument type is not relevant and applicable here.

As in MPI, our interface also provides non-blocking versions of the collective operations.

# 7. EXPERIMENTS AND BENCHMARKS

We have implemented the interface as outlined in Section 6 and so far the basic, $\lceil s/k \rceil$-round direct algorithms for the collective operations (using non-blocking send and receive operations). We therefore do not expect any significant improvements in communication performance over current MPI library implementations. Instead, our initial goal is to assess the costs of setting up isomorphic neighborhoods, where we do expect significant savings over the creation of distributed graph communicators. We also take first steps towards systematic benchmarking of sparse, collective communication operations in general. We discuss various experiment setups, which we benchmark both with `MPI_Neighbor_alltoallw` and `Iso_neighbor_alltoallw`. Finally, we present benchmarking results of the stencil examples discussed in Section 2.

All experiments have been carried out on a small, 36-node InfiniBand cluster. Each node is built from two 8-core AMD Opteron 6134 processors running at 2.3 GHz and 32 GiB of main memory. The operating system is Linux 2.6.32-504.8.1.el6.x86_64. The used compiler is gcc 4.4.7 20120313. Three MPI implementations, namely NEC MPI 1.3.1, MVAPICH2 2.1 and Open MPI 1.8.4 have been used.

We measure the execution time using `MPI_Wtime`, despite its possibly limited precision [11]. Processes are synchronized between measurements with our own external implementation of a dissemination barrier [19], since relying on a library-specific implementation of `MPI_Barrier` may impact the comparability of the results.

We refer to the measurement of the duration of an operation for specific input parameters and environment like number of MPI processes as an *experiment*. Experiments with different parameters are carried out in a random series, that includes multiple executions of each experiment. Each series is distributed over a number of `mpirun` calls, since the execution context can strongly influence the results [12]. Outliers are removed using Tukey's method with a factor of

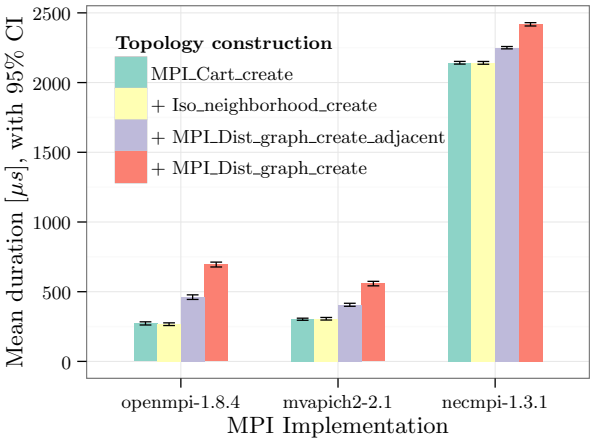


**Figure 2: Time for creating a 2D *von Neumann* neighborhood of radius 1 on top of `MPI_COMM_WORLD`, periodic in all dimensions, row major order of neighbors list, 30 nodes, 16 processes per node ($24 \times 20$ virtual torus).**

3.0, see, e.g., [6]. We report the mean times of the remaining data with their 95% confidence intervals (CI).

## 7.1 Neighborhood creation overheads

We first measure the overheads for setting up communicators and neighborhoods. In Figure 2 we compare the times for creating a Cartesian communicator (e.g., for collective communication in a simple von Neumann neighborhood) and for setting up the von Neumann neighborhood using the `MPI_Dist_graph_create_adjacent`, `MPI_Dist_graph_create` and `Iso_neighborhood_create` functions, including the creation of the underlying Cartesian communicator. Here and in Figure 3 the benchmark was executed 5 times (5 `mpirun`s), and in every execution each experiment was repeated 10 times. As can be seen, the `Iso_neighborhood_create` is completely dominated by the creation of the Cartesian communicator, which means that if the communicator is already given the cost of attaching and processing the neighborhood information is insignificant. As expected, `MPI_Dist_graph_create` is the most expensive of the topology creation functions, as seen even more clearly in Figure 3 where we measure only the raw communicator creation time. This clearly shows that `MPI_Dist_graph_create_adjacent` should be used where possible, as a performance guideline would also suggest [21]. In Figure 3 we create the communicators for a larger, radius 3 Moore neighborhood using the MPI functionality as well as our `Iso_neighborhood_create` function. It is interesting to observe that the setup times differ significantly between the three MPI libraries, and also that their relative order change completely in the two cases. In Figure 2 the NEC library is the slowest by far, but the fastest for the larger neighborhood in Figure 3. In both cases we ran with 30 nodes and 16 MPI processes per node, and a $24 \times 20$ Cartesian topology. For the creation of the Cartesian communicator, the `reorder` flag was set; for the creation of the neighborhood topologies on top of this communicator, no reordering was set, so as to make the MPI calls consistent with `Iso_neighborhood_create` that does not have a

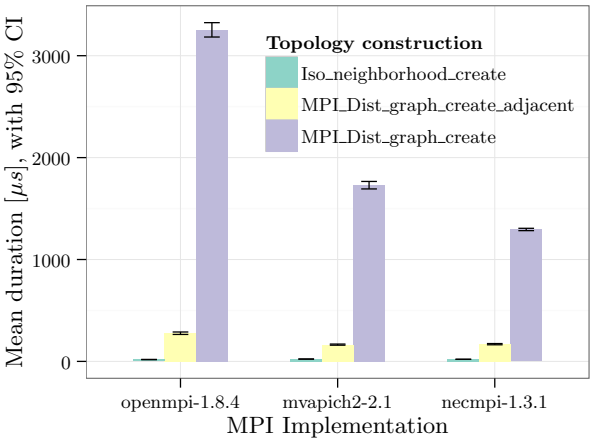


**Figure 3: Time for creating a 2D *Moore* neighborhood of radius 3 on top of a Cartesian communicator, periodic in all dimensions, row major order of neighbors list, 30 nodes, 16 processes per node ($24 \times 20$ virtual torus).**

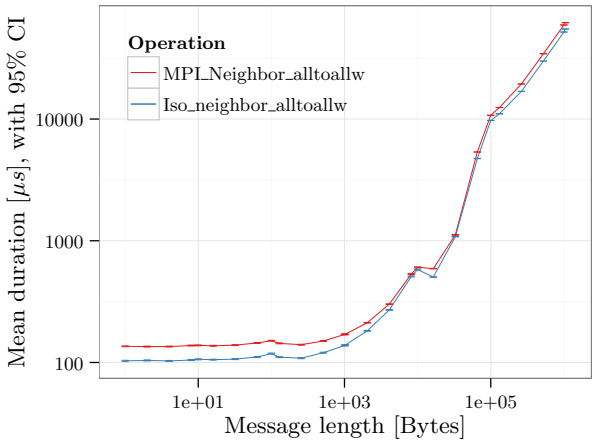


**Figure 4: Time needed by `MPI_Neighbor_alltoallw` and `Iso_neighbor_alltoallw` in a 2D *Moore* neighborhood of radius 3, periodic in all dimensions, row major order of neighbors list, 30 nodes, 1 process per node ($6 \times 5$ virtual torus), MVAPICH2 2.1.**

`reorder` parameter.

Our benchmark makes it possible to specify more complex generalizations of the von Neumann and Moore neighborhoods. For a given, $d$-dimensional Cartesian communicator which is periodic in the first $t \leq d$ dimensions, a generalized, $d$-dimensional, radius-$k$ Moore neighborhood consists of all relative neighbors where all coordinates are less than or equal to $k$ (Chebyshev distance). A generalized radius-$k$ von Neumann neighborhood consists of all neighbors with Manhattan distance at most $k$. A given neighborhood can be further refined by subtracting a smaller, radius-$k'$, $k' < k$ neighborhood. To be able to investigate how the structure of the neighborhood influences creation and communication performance, it is also possible to randomize the order of the neighbors lists, either using the same random order for all processes or with a different order for each process. Finally, it is possible to take out the $n$ first neighbors from the lists. Our translation interface from relative coordinates to process ranks provides helpful functionality for the benchmark implementation. For $p$ given MPI processes, the factorization for a $d$-dimensional Cartesian communicator can either be chosen by the `MPI_Dims_create` function, or specified explicitly by a list of dimension sizes. During our experiments we observed that different MPI libraries can give different, sometimes peculiar factorizations for the same $p$ [22].

We think that this structured experimental setup allows a meaningful assessment of the performance of neighborhood collectives in somewhat application relevant scenarios. In the extreme it allows to compare neighborhood collective performance to the performance of global collectives like `MPI_Alltoallv`, by specifying full neighborhoods, and thereby to assess performance guidelines that `MPI_Alltoallv` shall perform better in such cases [21]. Or, conversely, to check that for small, constant sized neighborhoods, neighborhood collectives can indeed perform better than an awkward, global collective.

## 7.2 Neighborhood communication

To measure the raw performance of the neighborhood collectives, excluding time to set up communicators and neighborhoods, we benchmark the situation where blocks of the same, basic type in contiguous buffers are exchanged. We can use both Moore and von Neumann neighborhoods as explained above. We compare `MPI_Neighbor_alltoallw` to `Iso_neighbor_alltoallw` for different numbers of MPI processes. Block sizes (given in Bytes) are varied for each fixed setup. The dimensions for the Cartesian communicator were given explicitly in order to circumvent the different behaviors of `MPI_Dims_create`. The benchmark was executed 5 times, and in every execution each experiment was repeated 100 times.

Figures 4 and 5 compare `Iso_neighbor_alltoallw` to `MPI_Neighbor_alltoallw`. In Figure 4 we use a radius-3 Moore neighborhood, and therefore `MPI_Neighbor_alltoallw` is used on a topology created with `MPI_Dist_graph_create_adjacent`. Here our interface and algorithm performs significantly better for every message size compared to the MPI operation. This is not always the case, though, but we found no experiments (with any of the MPI libraries we used) where our algorithm performs much slower than the MPI operation; durations are always in the same ballpark. In Figure 5, where we use a radius-1 von Neumann neighborhood, our algorithm can be compared to `MPI_Neighbor_alltoallw` on a bare Cartesian topology. For some message sizes, there are statistically significant differences between both.

Figure 6 compares the three MPI libraries. We use our `Iso_neighbor_alltoallw` with a radius-1 Moore neighborhood. As can be seen, the MPI libraries differ significantly, with some quite erratic behavior in certain message ranges with Open MPI 1.8.4. For small and large message sizes, the NEC library performs the best.

Finally, in Figure 7 we investigate the possible influence that the order of the neighbors may have on communication performance. With a 2D radius-3 Moore neighborhood, we can give the neighbors in either row or column major order to the `Iso_neighborhood_create` call (a random ordering

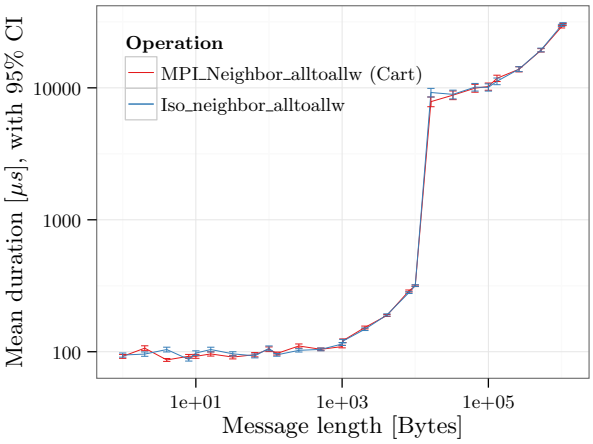


**Figure 5: Time needed by `MPI_Neighbor_alltoallw` on a Cartesian topology and by `Iso_neighbor_alltoallw` in a 2D *von Neumann* neighborhood of radius 1, non-periodic in all dimensions, row major order of neighbors list, 30 nodes, 16 process per node ($24 \times 20$ virtual mesh), Open MPI 1.8.4.**

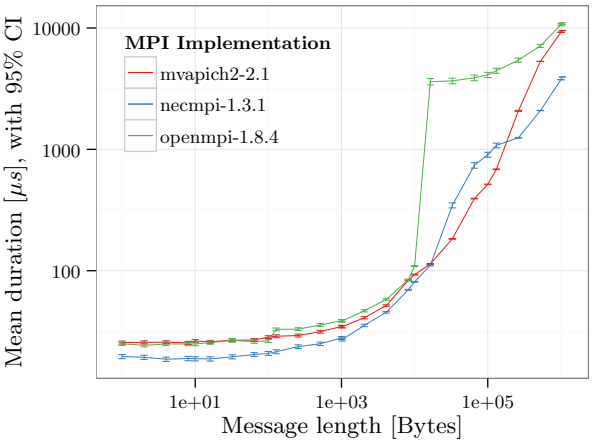


**Figure 6: Time needed by `Iso_neighbor_alltoallw` in a 2D *Moore* neighborhood of radius 1, periodic in all dimensions, row major order of neighbors list, 30 nodes, 1 process per node ($6 \times 5$ virtual torus), different MPI implementations.**

is also possible, but we do not report results here, because they do not differ significantly). In the message size range between $2^7$ B and $10^4$ B column major is faster than row major. Almost everywhere else, there is hardly a difference. It would be interesting to investigate the influence of the neighbor orders on the MPI collectives, and our benchmark makes this possible.

## 7.3 Stencil communication

In order to measure the performance of the sparse alltoall communication operations in context and with different amount and structure of data between different neighbors, we have implemented the two stencil-patterns discussed in Section 2, both with using `MPI_Neighbor_alltoallw` and our own `Iso_neighbor_alltoallw`, but without any actual stencil update. This kernel benchmark was executed 5 times and in each execution the experiment was repeated 100 times. Execution times include the time to set up the neighborhoods. Arbitrary depth-$k$ halos can be used, with a triangular pattern needed in the 5-point stencil case when $k > 1$. The order of the per process matrix can be varied, the element basetype is `MPI_BYTE`.

Figure 8 and Figure 9 give results for the MVAPICH2 2.1 and NEC MPI 1.3.1 libraries. In Figure 8 we use matrix order $n = 10^4$ with a large halo of depth 10, and small numbers of processes. We observe that our isomorphic interface implementation is significantly faster (in the statistical sense) than the MPI neighborhood collective implementation by a few percent. However, for NEC MPI 1.3.1 this is not always the case. As can be seen, performance is heavily dependent on the sizes of the dimensions in the Cartesian setup. This seems due to the different amount of communication needed for the different factorizations.

In Figure 9 we show results for a Moore neighborhood (9-point stencil) for larger numbers of MPI processes, here with smaller matrices of order $10^2$ elements and a halo of depth 2. Again, we observe the implementation with `Iso_neighbor_alltoallw` to be faster.

# 8. DISCUSSION: MPI STANDARD

Part of the motivation for the proposed interface was the lacking functionality for more complex, stencil-like, sparse collective communication on Cartesian MPI communicators. As mentioned several times, the MPI 3.1 standard treats Cartesian and the general, distributed graph communicators quite differently. For instance, it is not possible to associate weights with the neighbors in Cartesian communicators, and the creator function does also not take an `MPI_Info` parameter. This makes a difference in the capabilities that an MPI implementation will have in mapping the virtual topology to the actual communication network. There is missing functionality for explicitly querying the (implicitly defined) neighbors in a Cartesian communicator. In non-periodic Cartesian topologies some neighbors are inevitably `MPI_PROC_NULL`, and in the neighborhood collectives the corresponding buffers are not significant, but nevertheless have to be specified and accounted for in the application. For graph topologies, the MPI standard implies that neighbors must be actual ranks (that is, `MPI_PROC_NULL` is excluded)[2], and any, possibly convenient `MPI_PROC_NULL` neighbors must therefore be eliminated from the rank lists used in `MPI_Dist_graph_create_adjacent`. Either way, the application code will look different, depending on whether the underlying communicator for the neighborhood collective communication is Cartesian or a graph. This is unfortunate.

For the `MPI_Cart_rank` translation function, it is illegal to give a coordinate that is out of range in a non-periodic dimension. For `MPI_Cart_shift` instead `MPI_PROC_NULL` is returned in such a case. It is not obvious what the reason for this asymmetry is.

We argued that the most useful interfaces for sparse collective communication will need full flexibility in describing the

[2]The standard is not completely clear, and MPI libraries were found to behave differently!

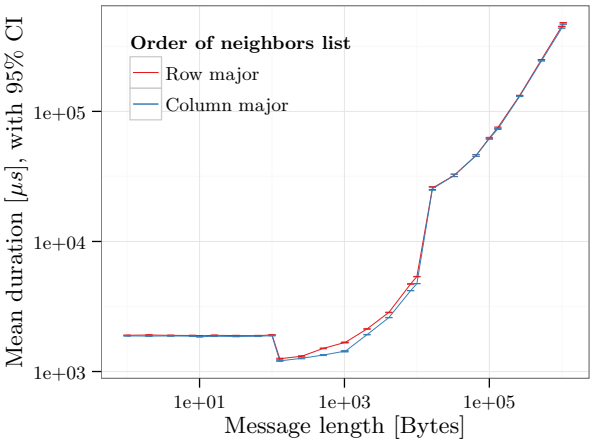


**Figure 7: Time needed by `Iso_neighbor_alltoallw` in a 2D *Moore* neighborhood of radius 3, non-periodic in all dimensions, 30 nodes, 16 process per node ($24 \times 20$ virtual mesh), Open MPI 1.8.4, list of neighbors in different orders.**

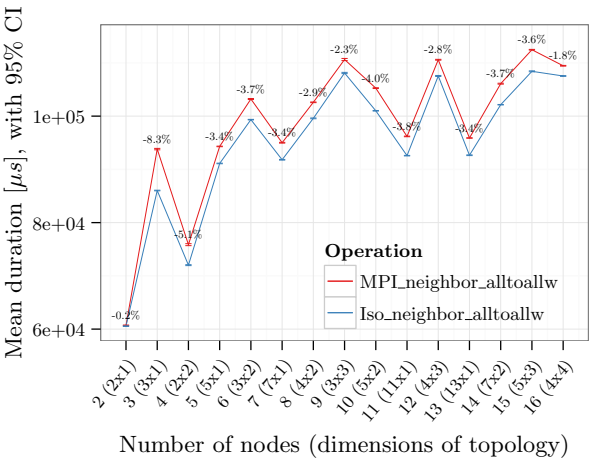


**Figure 8: Time required by stencil kernel, $10^4 \times 10^4$ matrix, *von Neumann* shaped stencil, halo depth 10, one process per node, MVAPICH2 2.1, different numbers of nodes. Improvement of `Iso_neighbor_alltoallw` over `MPI_Neighbor_alltoallw` annotated in percent.**

structure of the communication buffers, and thus should be of the `-w`-variety (or better, see [23]). The MPI 3.1 standard lacks an `MPI_Neighbor_allgatherw` function. MPI 3.1 also does not provide sparse reduction functions, as we propose here.

We showed that isomorphic neighborhoods make it easier to make algorithmic decisions. It would be a possibility to assert this to the current MPI 3.1 functions during creation of distributed graph communicators by providing a corresponding `MPI_Info` value. The difficulty here lies in defining what isomorphic means; in our proposal it is given meaning relative to an underlying Cartesian topology.

In our stencil-implementation we found use for a triangular datatype constructor. There is no such constructor in MPI, instead the pattern had to be (space) inefficiently described using an `MPI_Type_indexed` constructor. In our library we provide the following datatype constructor for two-dimensional triangular layouts:

```
Type_create_triangular(int count,
  int firstblock, int blockincrement,
  int stride, int strideincrement,
  MPI_Datatype oldtype, MPI_Datatype *newtype)
```

This describes a layout of `count` regularly changing blocks. The number of elements in the $i$th block is `firstblock` + $i \times$ `blockincrement`, and the $i$th block is placed at offset $i \times$ (`stride` + `strideincrement`). All of `blockincrement`, `stride` and `strideincrement` can be negative. With this constructor, it is possible to describe all the triangles shown in Figure 1. Implemented as an indexed type, the constructor is space inefficient, with space proportional to `count` instead of constant. This type of layout could presumably also be handled more efficiently by the datatype engine than the descriptively equivalent implementation with `MPI_Type_indexed`.

## 9. CONCLUDING REMARKS

MPI 3.0 introduced promising, new functionality for sparse collective communication. Sparse process neighborhoods in MPI can be either highly structured, but severely limited as implicitly defined by Cartesian communicators, or defined by fully general communication graphs, which must then be analyzed by the MPI library in order to select the most appropriate communication algorithm. In this paper we investigated a middle ground which brings needed expressiveness to Cartesian communicators for sparse collective communication, while having low setup and analysis overhead in selecting good communication algorithms. We showed that the simple, isomorphic, sparse collective interface allows for (much) faster setup times than the graph topology constructors of MPI, even when including the time to create the underlying, Cartesian communicator. This opens possibilities for using sparse, collective communication operations in more dynamic settings where neighborhoods change frequently. We sketched some algorithmic approaches to implementing isomorphic, sparse collectives, all benefitting from the fact that processes can assume that they have identical, relative neighborhoods. Only the basic algorithm was implemented, though; nevertheless, its communication performance is on par with current MPI library implementations of the neighborhood collectives. We used our interface discussion as a vehicle to develop a benchmark for MPI-like, sparse collective communication. First results were given here. It will be interesting to see what can be further gained by zero-copy implementations of the message-combining algorithms outlined in Section 5.

The idea of specifying neighborhoods by relative coordinates in a given mesh or torus layout could also be used for non-isomorphic neighborhoods. This could provide convenient helper functionality, but the advantages for the computation of communication schedules would be lost. We did not explore this extension here.

## Acknowledgments

The authors want to thank members of the EPiGRAM consortium, especially Dan Holmes and Mirko Rahn, for constructive discussions that led to a number of clarifications of

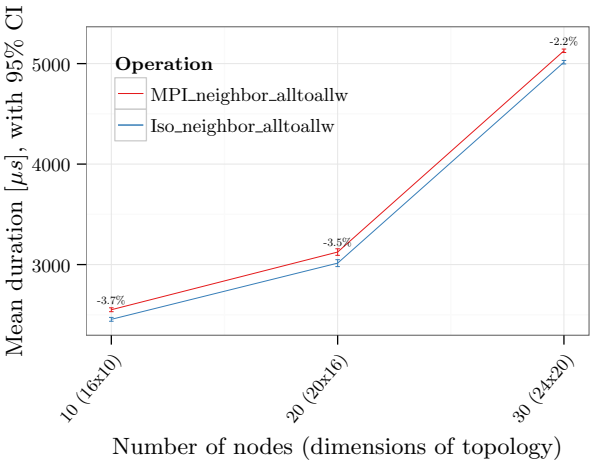


**Figure 9: Time required by stencil kernel, $10^2 \times 10^2$ matrix, *Moore* shaped stencil, halo depth 2, 16 processes per node, NEC MPI 1.3.1, different numbers of nodes, improvement of `Iso_neighbor_alltoallw` over `MPI_Neighbor_alltoallw` annotated in percent.**

the concepts discussed in this paper.

## 10. REFERENCES


[1] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, T. Hoefler, S. Kumar, E. Lusk, R. Thakur, and J. L. Träff. MPI on millions of cores. *Parallel Processing Letters*, 21(1):45–60, 2011.

[2] J. Bruck, C.-T. Ho, S. Kipnis, E. Upfal, and D. Weathersby. Efficient algorithms for all-to-all communications in multiport message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 8(11):1143–1156, 1997.

[3] W. Gropp, T. Hoefler, R. Thakur, and E. Lusk. *Using Advanced MPI*. MIT Press, 2014.

[4] W. Gropp and E. L. Lusk. Reproducible measurements of MPI performance characteristics. In *EuroPVM/MPI*, pages 11–18, 1999.

[5] G. Hager and G. Wellein. *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press, 2011.

[6] J. Hedderich and L. Sachs. *Angewandte Statistik*. Springer, 14 edition, 2012.

[7] T. Hoefler and S. Gottlieb. Parallel zero-copy algorithms for fast fourier transform and conjugate gradient using MPI datatypes. In *Recent Advances in Message Passing Interface. 17th European MPI Users' Group Meeting*, volume 6305 of *Lecture Notes in Computer Science*, pages 132–141. Springer, 2010.

[8] T. Hoefler, A. Lumsdaine, and W. Rehm. Implementation and performance analysis of non-blocking collective operations for MPI. In *Proceedings of the ACM/IEEE Conference on High Performance Networking and Computing (SC)*, page 52, 2007.

[9] T. Hoefler, R. Rabenseifner, H. Ritzdorf, B. R. de Supinski, R. Thakur, and J. L. Träff. The scalable process topology interface of MPI 2.2. *Concurrency and Computation: Practice and Experience*, 23:293–310, 2011.

[10] T. Hoefler and T. Schneider. Optimization principles for collective neighborhood communications. In *IEEE/ACM Conference on High Performance Computing Networking, Storage and Analysis (SC)*, page 98, 2012.

[11] T. Hoefler, T. Schneider, and A. Lumsdaine. Accurately measuring overhead, communication time and progression of blocking and nonblocking collective operations at massive scale. *International Journal of Parallel, Emergent and Distributed Systems*, 25(4):241–258, 2010.

[12] S. Hunold, A. Carpen-Amarie, and J. L. Träff. Reproducible MPI micro-benchmarking isn't as easy as you think. In *Recent Advances in Message Passing Interface. (EuroMPI/ASIA)*, pages 69–76, 2014.

[13] Intel(R) MPI Benchmarks. http://software.intel.com/en-us/articles/intel-mpi-benchmarks.

[14] K. C. Kandalla, A. Buluç, H. Subramoni, K. Tomko, J. Vienne, L. Oliker, and D. K. Panda. Can network-offload based non-blocking neighborhood MPI collectives improve communication overheads of irregular graph algorithms? In *International Conference on Cluster Computing Workshops, (CLUSTER Workshops)*, pages 222–230, 2012.

[15] S. Kumar, P. Heidelberger, D. Chen, and M. L. Hines. Optimization of applications with non-blocking neighborhood collectives via multisends on the Blue Gene/P supercomputer. In *24th IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1–11, 2010.

[16] MPI Forum. *MPI: A Message-Passing Interface Standard. Version 3.1*, June 4th 2015. www.mpi-forum.org.

[17] OSU MPI benchmarks. http://mvapich.cse.ohio-state.edu/benchmarks/.

[18] R. Reussner, P. Sanders, and J. L. Träff. SKaMPI: a comprehensive benchmark for public benchmarking of MPI. *Scientific Programming*, 10(1):55–65, 2002.

[19] G. Taubenfeld. *Synchronization Algorithms and Concurrent Programming*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[20] T. Toffoli and N. Margolus. *Cellular Automata Machines: A New Environment for Modeling*. MIT Press, 1987.

[21] J. L. Träff, W. D. Gropp, and R. Thakur. Self-consistent MPI performance guidelines. *IEEE Transactions on Parallel and Distributed Systems*, 21(5):698–709, 2010.

[22] J. L. Träff and F. D. Lübbe. Specification guideline violations by `mpi_dims_create`. In *Recent Advances in the Message Passing Interface (EuroMPI)*, 2015. To appear.

[23] J. L. Träff and A. Rougier. Zero-copy, hierarchical gather is not possible with MPI datatypes and collectives. In *Recent Advances in Message Passing Interface. (EuroMPI/ASIA)*, pages 39–44, 2014.

[24] J. L. Träff, A. Rougier, and S. Hunold. Implementing a classic: Zero-copy all-to-all communication with MPI datatypes. In *28th ACM International Conference on Supercomputing (ICS)*, pages 135–144, 2014.